DATA+AI SUMMIT 2022

Building a ML enrichment framework using advanced delta table features

Peter Vasko
Data Architect, Emplifi

ORGANIZED BY databricks

about the speaker

Peter

- studied computer science
- 14 years of data-engineering
- last 7 years with big data technologies
- leading a team of 8 talented data engineers
- responsibilities of the team
  - building a PB scale data-lake from scratch
  - deploying (open-source) technologies on top of data-lake
  - running ML in production
  - supporting ML, analytics and product

our setup

our social media specific environment

- continuous flow of records from social media networks
  - 350M+ inserts / updates per day, 50k+ per second in peaks

social media record (post, comment, tweet, ...)

id, author, **message**, **attachments** [urls],
common metadata (created, updated, ...),
network specific data (retweets, likes, views, ...)

- NLP and computer vision models
  - NLP -> sentiment, topics, named entities, aspects, ...
  - computer vision -> objects, logos, similarities, NSFW, ...

aws

databricks

mlflow

motivation

existing spark solution was opaque and inefficient

- no clear answers to some typical questions
- inefficiencies on multiple levels
- no standards
  - hard to maintain individual jobs
  - barrier for other teams

overview

what we wanted to build

- an end-to-end framework usable by all data teams
- able to evaluate potentially hundreds of rules
- process each enrichment efficiently and according to the use-case
- accommodate both streaming and batch

enrichment

output of ML model appended to the original record

rule

enrichment specific set of conditions to be evaluated on record level, they determine if ML model should be applied or not

new solution

criteria

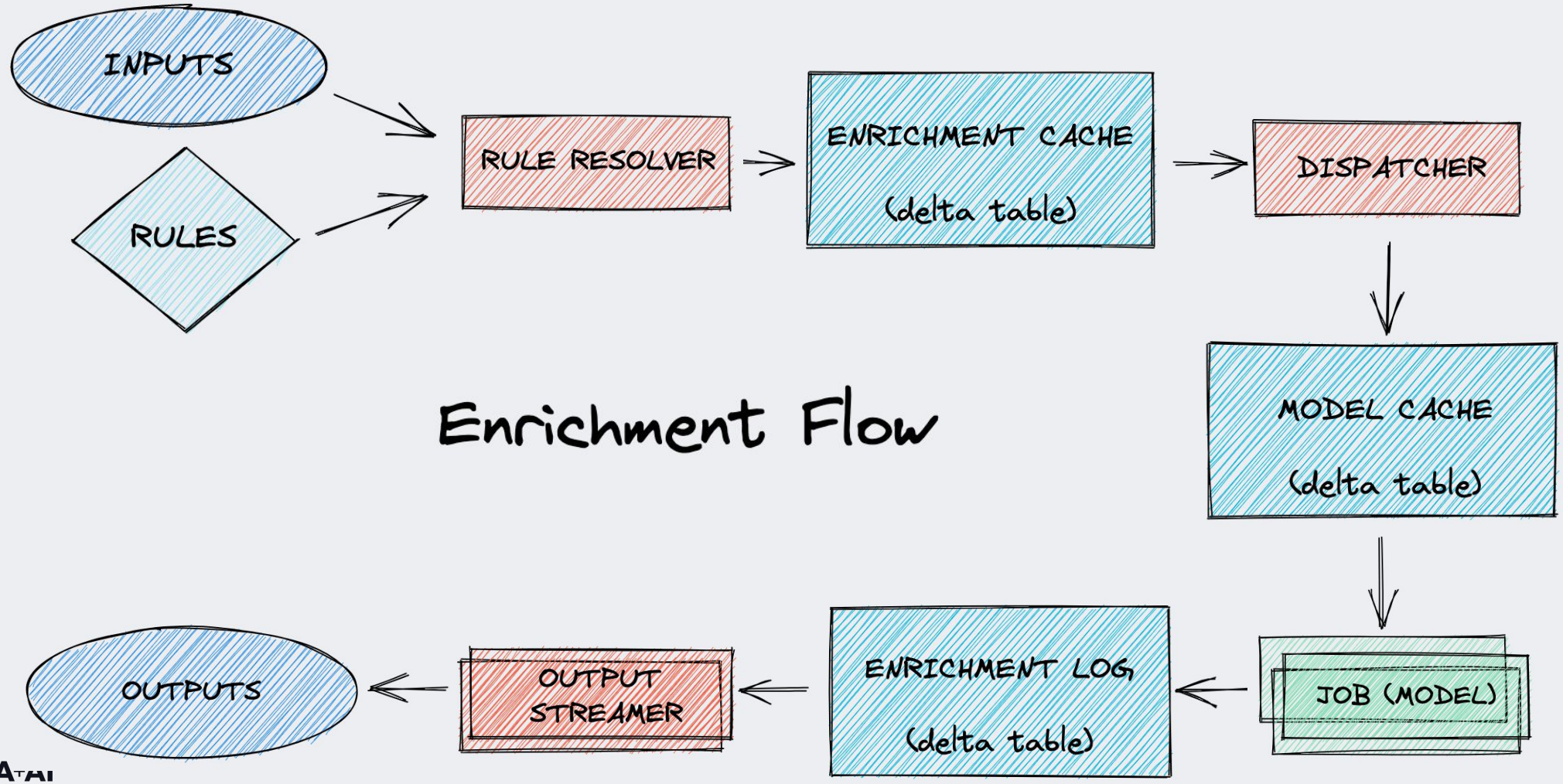
- transparent
- efficient
- scalable
- standardized and open to other teams

enrichment flow

high-level and components

high-level flow

from input to final destination



components | rule resolver

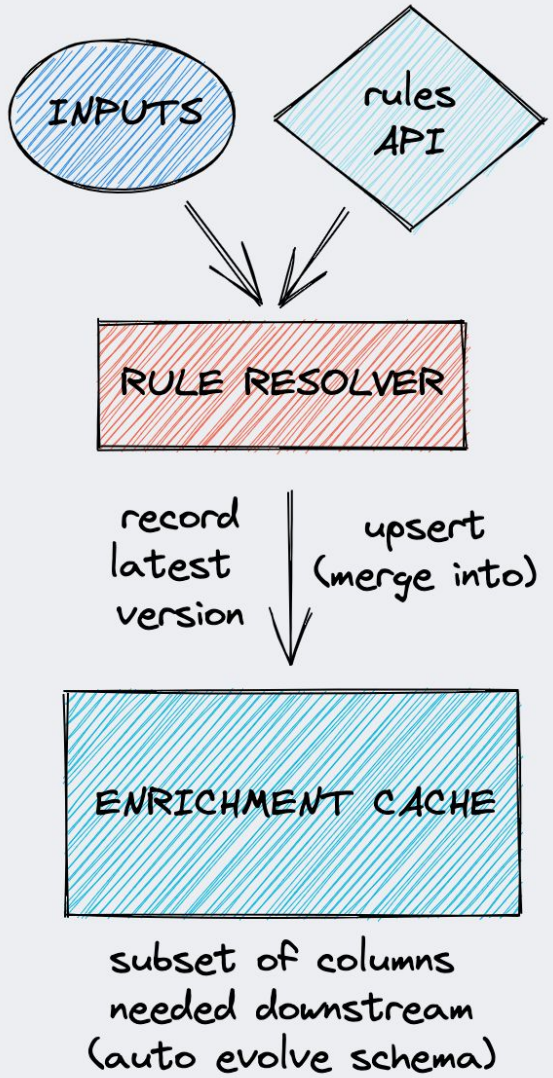
evaluate and keep only what is needed

- **filter out** everything we don't need
- resolve all the rules on record level -> save results into delta table
- M:N relation between rules and records
- **enrichment cache** – selective **merge into**
  - common subset of columns, each record only once (upsert)
  - info about relevant models on record level
  - delta features
    - auto schema evolution (delta.io v0.8.0+)
    - auto optimize (DBR 10.4+)
    - delta change data feed (DBR 8.4+)

example of **rule**

IF source social network in ["XYZ", "ABC"]
& has image / video attachments
& internal rating in ["A+", "A"]

THEN apply model X



components | rule resolver

selective merge into with pyspark

```
# selective MERGE INTO enrichment_cache delta table
# only actually updated records, i.e. higher version are updated
# new records are inserted

from delta.tables import DeltaTable

base = DeltaTable.forName(spark, 'enrichment_cache').alias('ec')
base.merge(
   updates_df.alias('u'),
   'ec.id = u.id'
 ).whenMatchedUpdateAll('u.version > ec.version').whenNotMatchedInsertAll().execute()
```

components | rule resolver

delta features

```
# enable features with spark.conf
spark.conf.set('spark.databricks.delta.schema.autoMerge.enabled', True)
spark.conf.set('spark.databricks.delta.optimizeWrite.enabled', True)

# enable change data feed and auto optimize with write at table creation
(
    df.write
    .option('mergeSchema', 'true')
    .option('delta.autoOptimize.optimizeWrite', 'true')
    .option('delta.enableChangeDataFeed', 'true')
    .saveAsTable(ENRICHMENT_CACHE_TABLE)
)

# or later with SQL
ALTER TABLE ENRICHMENT_CACHE_TABLE SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

components | dispatcher

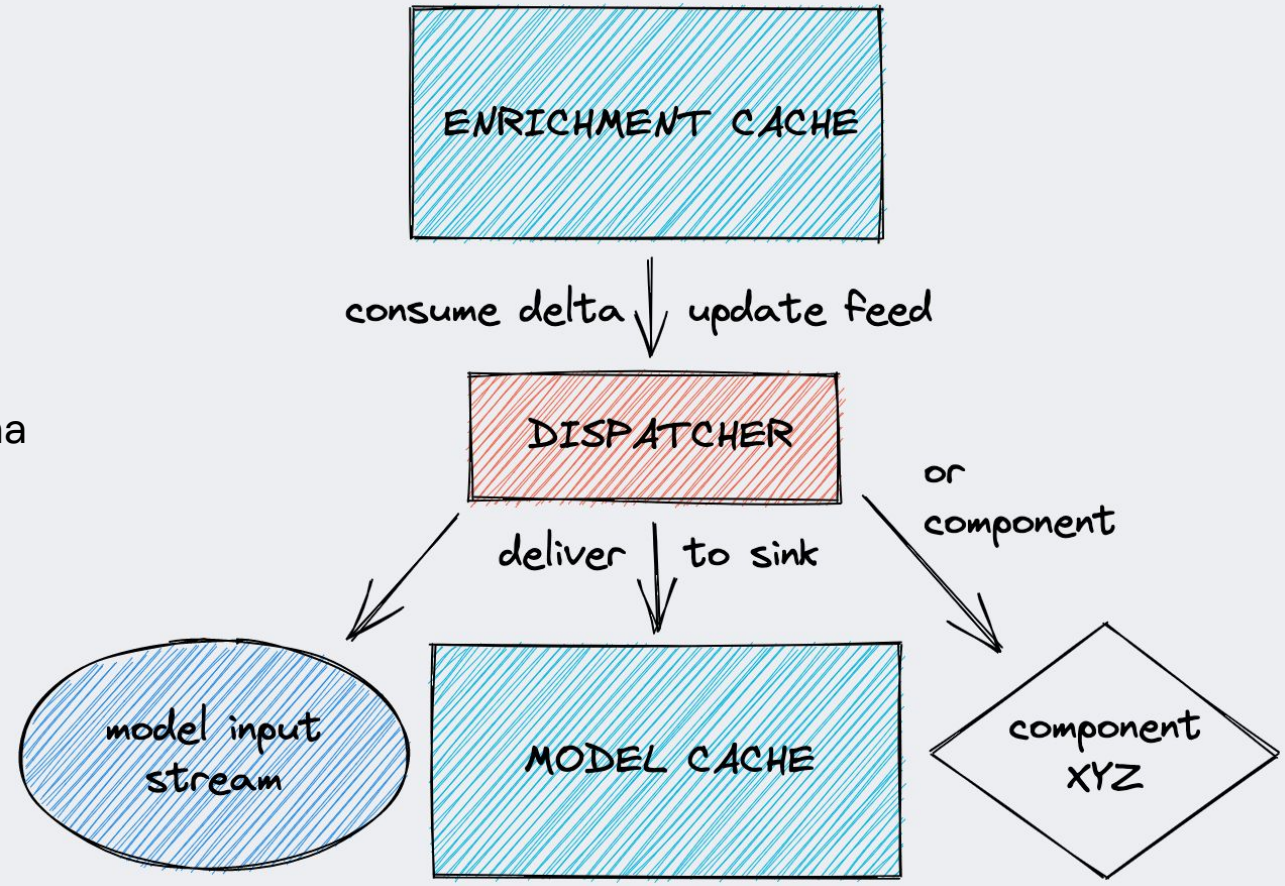
deliver relevant data for processing

- on input only records want to enrich
- send records down various **sinks**
- processing delta **change feed** -> big timesaver
- custom components
- output (for DeltaSink) – **model cache**
  - **one record per job** (model version)
  - subset of job specific fields (as json) + schema
  - partitioning by job

job (enrichment specific)

spark job (streaming / batch) processing only relevant records to the job with enrichment specific ML model

e.g. adding sentiment to a record from social network by processing its message column



components | dispatcher

deliver records to various sinks

```python
class Sink(ABC):
    def __init__(self, transform_batch: Callable[[DataFrame], DataFrame]):
        self.transform_batch = transform_batch

    @property
    @abstractmethod
    def name(self):
        pass

    @abstractmethod
    def _flush(self, df: DataFrame) -> None:
        pass

    def flush(self, df: DataFrame) -> None:
        self._flush(self.transform_batch(df))
```

components | dispatcher

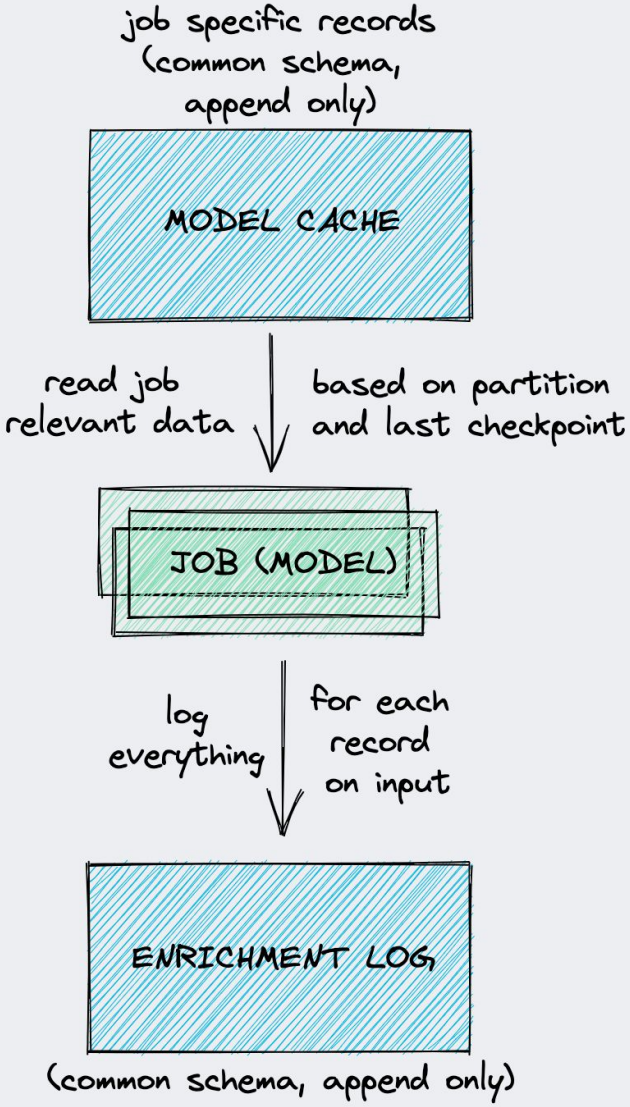
stream from delta feed

```
(
    spark
    .readStream
    .format('delta')
    .option('maxBytesPerTrigger', INPUT_STREAM_MAX_BYTES_PER_TRIGGER)
    .option('readChangeFeed', 'true')
    .table(ENRICHMENT_CACHE_TABLE)
    .filter(f.col('_change_type').isin('insert', 'update_postimage'))
    .writeStream
    .queryName(QUERY_NAME)
    .option('checkpointLocation', QUERY_CHECKPOINT)
    .trigger(processingTime='10 seconds')
    .foreachBatch(process_batch)
    .start()
)
```

components | jobs (model)

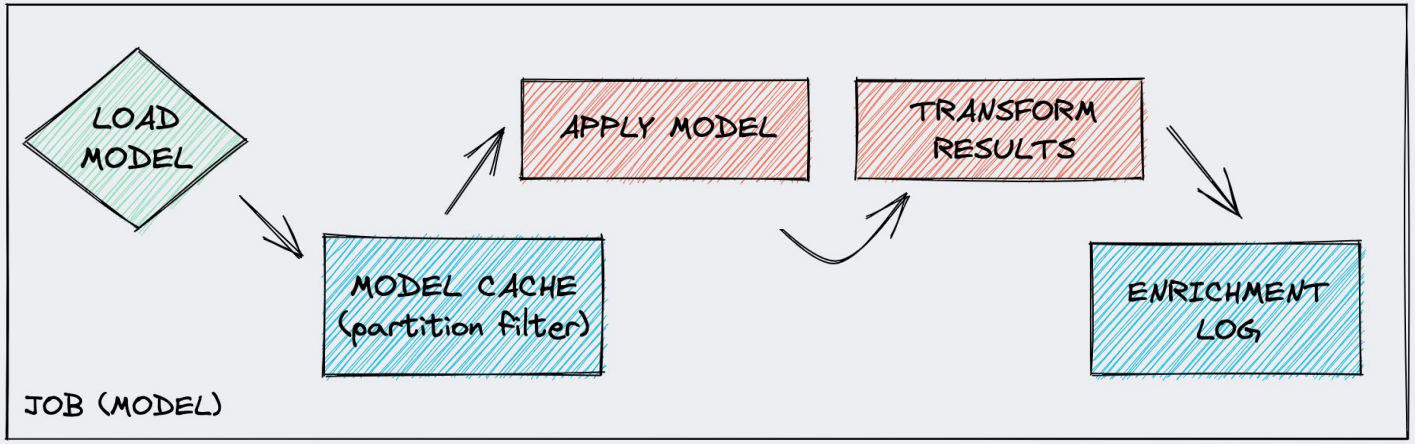
job specific data, standardized output

- jobs -> batch or stream processing
  - **common library** + specific model
  - inputs from model cache - only the **job specific partition**
- output - **enrichment log**
  - everything from input (also nulls, errs)
  - with model output, model version and payload
- atomic with **exactly once** processing
  - job processes everything from last checkpoint
  - writes result to enrichment log
  - dispatcher can anti-join on enrichment log

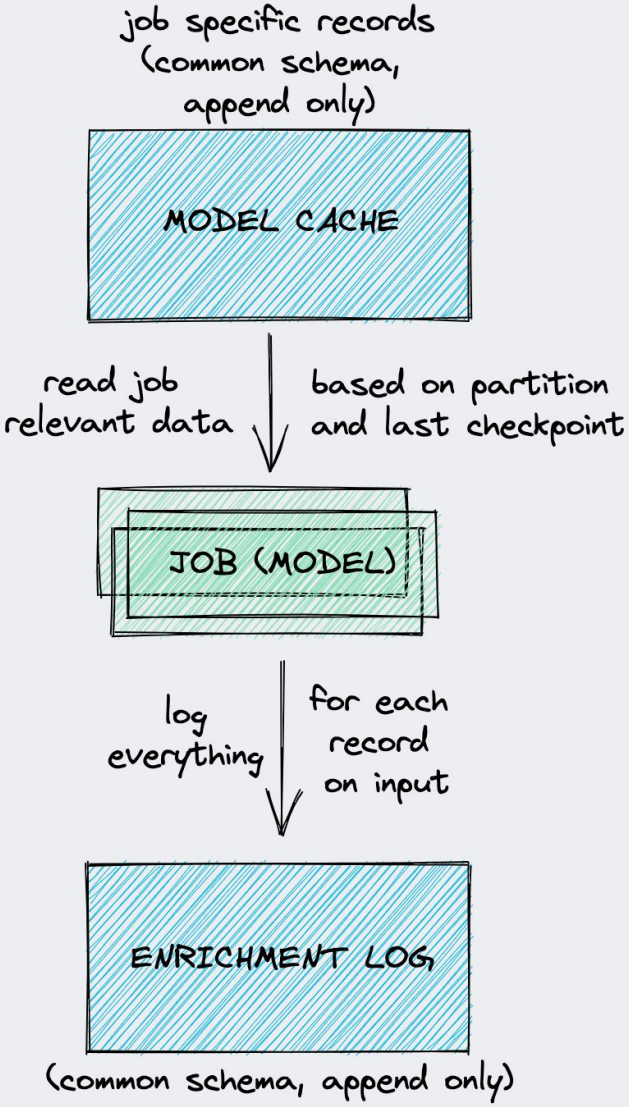


components | jobs (model)

enrichment job implementation flow



- job **developer friendly** flow
  - inputs ready
  - no need to care about downstream



components | jobs (model)

implement enrichment job from common library

```
def process_batch(df: DataFrame, batch_id: int) -> None:
    job = EnrichmentJob(...)

    job.run(
        data_source=...
        transformations=[
            job.apply_enrichers([
                ...
            ]),
            ...
        ],
        output=job.write_to_log,
        finalizers=...
    )
```

components | jobs (model)

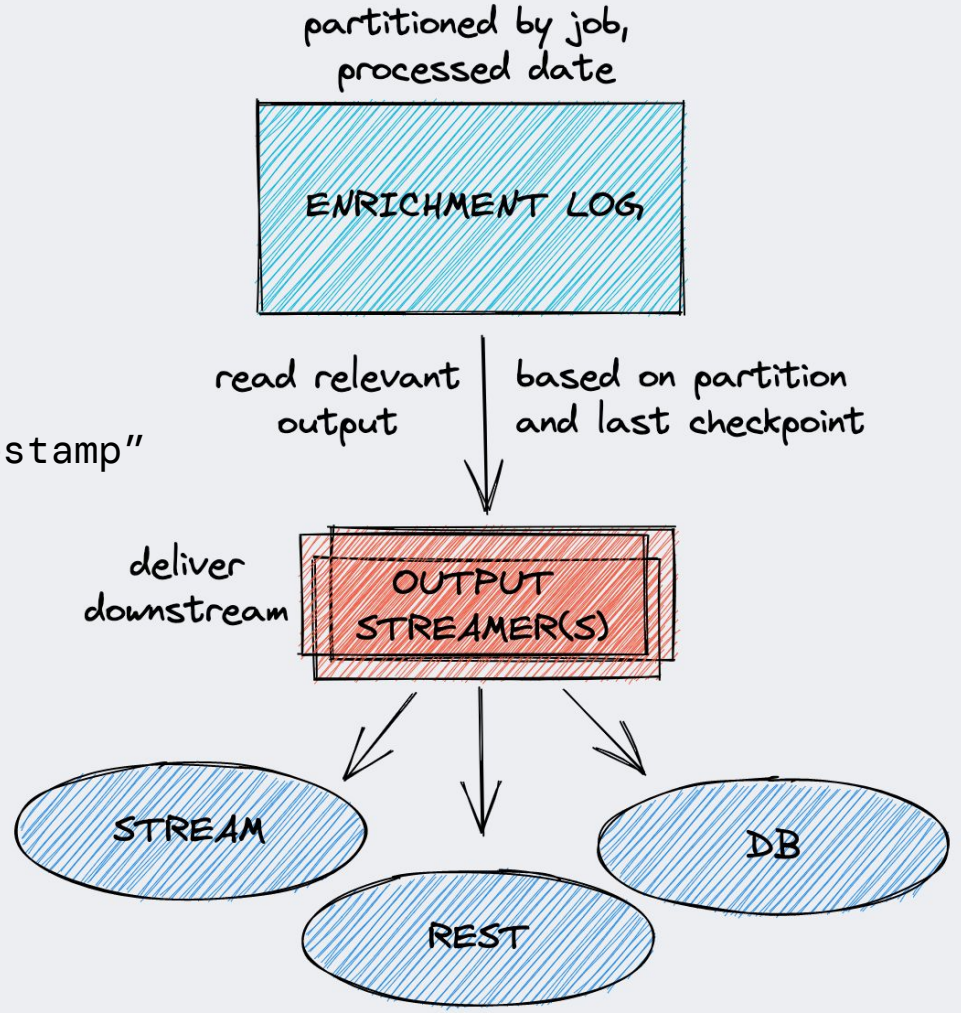
stream from delta input to delta output

```
(
    spark
    .readStream
    .format('delta')
    .table(MODEL_CACHE['table'])
    .filter(f.col('job') == f.lit(JOB))  # partitioning
    .repartition(spark.sparkContext.defaultParallelism)
    .writeStream
    .queryName(QUERY_NAME)
    .option('checkpointLocation', MODEL_CACHE_CHECKPOINT_PATH)
    .trigger(once=True)
    .foreachBatch(process_batch)
    .start()
)
```

components | output streamer

deliver data to downstream consumers

- various consumers / target systems
- sending data downstream efficiently
- **all** outputs **at one place** in delta
  - easy debug
  - same code for backfills (with new temporary job instance)
  - using delta options "`startingVersion`" or "`startingTimestamp`"



results and takeaways

takeaways

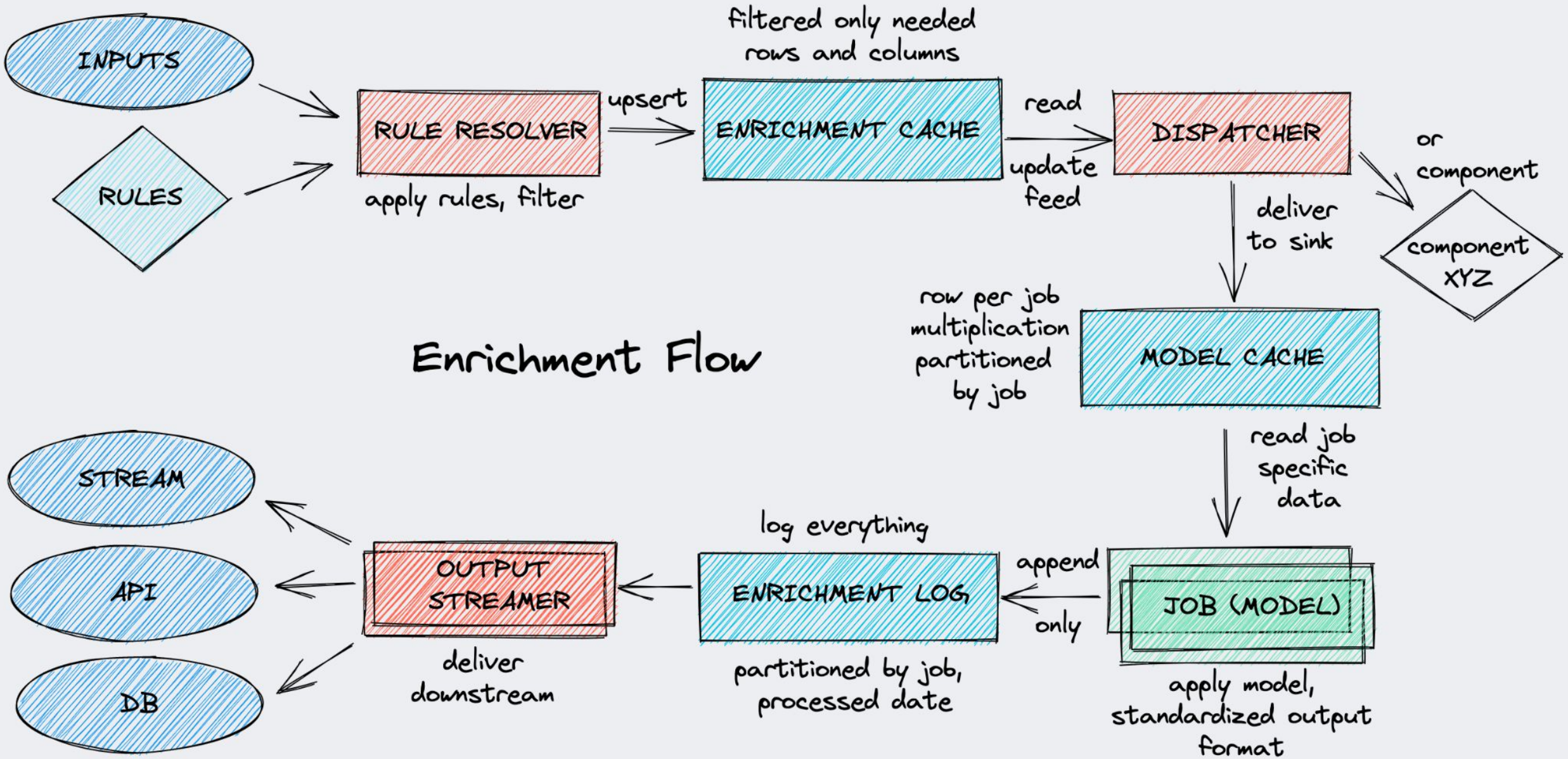
what we found out about the technologies

- delta
  - performing well in all of our scenarios
  - performance -> affected by size and structure
  - clever partitioning, compaction and optimize are needed
  - versioning -> easy backfills even in a streaming environment
- spark structure streaming
  - efficient (single-node clusters)
    - rule resolver -> `m5d.2xlarge`
    - dispatcher -> `m5d.xlarge`
    - output streamer -> `r5d.xlarge`
  - usable for both streaming and batch (checkpointing, `.foreachBatch()`)

result and added benefits

what we ended up building

- other teams now write efficient code applying models on records
- transparent == **trusted**
- **split of responsibility** within data teams
- **decoupling** parts of the enrichment process
  - **phased** and phases validated separately



Enrichment Flow

DATA+AI
SUMMIT 2022

Thank you

Peter Vasko
Data Architect | emplifi.io